Abstract—Local community search is an important research topic to support complex network data analysis in various scenarios like social networks, collaboration networks, and cellular networks. The evolution of networks over time has motivated several recent studies to identify local communities from dynamic networks. However, they only utilized the aggregation of disjoint structural information to measure the quality of communities, which ignores the reliability of communities in a continuous time interval. To fill this research gap, we propose a novel \((\theta, k)\)-core reliable community (CRC) model in the weighted dynamic networks, and define the problem of the most reliable community search that couples the desirable properties of connection strength, cohesive structure continuity, and the maximal member engagement. To solve this problem, we first develop an online CRC search algorithm by proposing a definition of eligible edge set and deriving the eligible edge set based pruning rules. After that, we devise a Weighted Core Forest-Index and index-based dynamic programming CRC search algorithm, which can prune a large number of insignificant intermediate results according to the maintained weight and structure information in the index, as well as the proposed upper bound properties. Finally, we conduct extensive experiments to verify the efficiency of our proposed algorithms and the effectiveness of our proposed community model on eight real datasets under different parameter settings.

Index Terms—Local Community Search, Reliable Community Search, Dynamic Networks

I. INTRODUCTION

Local community search has attracted much attention in recent years and has shown its great success in different applications, e.g., personalized recommendation \cite{1, 2}, destination marketing \cite{3}. Generally, local community search aims to identify a densely connected structure with regards to a query vertex. A number of works have been devoted to the problem of local community search in static networks. For instance, Clauset et al. in \cite{4} and Luo et al. in \cite{5} proposed local modularity to measure the quality of the local community. Luo et al. \cite{6} designed a random walk based method to detect communities from multiple networks. However, network data often changes over time. Therefore, there are also some studies of local community search in dynamic networks. For example, Bu et al. \cite{7} explored local community mining in distributed and dynamic networks from the multiagent perspective. It provides a modularity-based criterion to find local communities and update the communities in the dynamic network in an incremental manner by monitoring the changes. In another work, DiTursi et al. \cite{8} discovered the dynamic communities with optimal time intervals by minimizing the novel measurement called temporal conductance. The above works assess the quality of communities only based on their aggregated structural cohesiveness at independent timestamps, which tends to ignore the community continuity over time. In addition, they did not consider the edge weight, e.g., interaction or connection strength of vertices, which incurs the new computational challenge to solve local community search in dynamic networks.

To fill this research gap, we propose a novel community model of \((\theta, k)\)-core reliable community in dynamic networks where the community is a \(k\)-core with each edge weight no less than the weight threshold \(\theta\) and spans for a period of time. The most reliable local community search aims to find the community with the maximum reliability score, which is defined by coupling temporal continuity and member engagement. In other words, this work is to jointly consider three important properties of the community in the dynamic network, i.e., the connection strength, cohesiveness continuity, and member engagement.

Example 1. Figure 7 shows an illustrative example of a dynamic online social network of three timestamps, where the weight of an edge represents the connection strength. Assume that a cosmetics company wants to post ads for a new product. The effective way is to search a local community from an existing customer, where the community members might have continuous interaction or communication between each other with a certain strength. Our proposed \((\theta, k)\)-core reliable community can serve the application. Suppose we take \(v_0\) as the target user, \(k=3\), and \(\theta=0.4\) as the query input, we can identify a community \(C_3=\{v_0, v_2, v_3, v_4\}\) as the optimal result, which has four members and spans for two continuous timestamps \(t_1\) and \(t_2\). If we relax the structural requirement

* the corresponding author
from $k=3$ to $k=2$, while increase the interaction strength from $\theta=0.4$ to $\theta=0.5$, then there are two returned communities $C_1$ and $C_2$. It looks $C_2$ is better than $C_1$ for the company because the members of $C_2$ kept their close interaction for 3 continuous timestamps and the size of $C_2$ is not much less than that of $C_1$. This enables the company to identify ads placeholder easily and cost-effectively, and maximize the impact of their ads.

Although there are two similar existing works [9], [10] to identify meaningful communities over time, this work is still desirable due to the different research challenges. In [9], Li et al. defined the persistent community search as the maximal $k$-core where each vertex’s accumulated degree meets the $k$-core requirement within a time interval. They designed a novel temporal graph reduction algorithm and searched the maximum persistent community utilizing pruning and bounding techniques, which takes exponential complexity. In [10], Qin et al. proposed the stable communities by first selecting the centroid vertices where each centroid vertex has a certain number of neighbors with the desired similarity, and the star-shape of the centroid vertex and its neighbors exists frequently in a period of time; and then clustering the network vertices into stable groups based on the selected centroids. However, the persistent community in [9] did not consider the weight of the edge and its time complexity is too high for dealing with large-scale networks. The stable community in [10] ignored the continuity of the cohesive structure and failed to maximize member engagement.

To solve the proposed problem of the most reliable local community search, a naive idea is to enumerate all the possible community candidates and select the satisfied results by checking their edge weight and duration. But this will incur the exponential time cost. To address the computational challenge, in this paper, we firstly derive two important properties to prune the edge sets at different timestamps by analyzing the minimum edge requirement of $k$-core and calculating the reliability upper bound with regards to the eligible edge set. After that, we develop an efficient online search algorithm using the two properties, which can prune a large number of edge sets without probing their corresponding $(\theta,k)$-core community candidates. To further accelerate the query processing, we develop a weighted core forest index by maintaining the standard $\theta$-threshold values and the $(\theta,k)$-core structural information of vertices. This novel index can support the efficient retrieval of $k$-core with regards to different thresholds and timestamps. Following this, we design an index-based dynamic programming algorithm, which can reduce the computational cost by re-using the intermediate computational results. Furthermore, we derive the reliability upper bound of communities w.r.t. the time interval during the dynamic programming procedure to avoid probing the unsatisfactory community candidates. Besides, the index construction, maintenance, and compression are well presented in this paper.

The main contributions of this work are as below:

- We propose a novel problem of the most reliable community search that jointly considers community continuity, community size, and connection strength for online network analysis services.
- We develop an efficient online search algorithm by deriving and applying the properties of pruning the ineligible edges w.r.t. the given query conditions.
- We further present a weighted core forest index and develop an index-based dynamic programming algorithm to solve the most reliable community search problem in a more efficient way.
- We conduct extensive experiments to show the efficiency and effectiveness of the proposed algorithms and community model by using eight real-world datasets and comparing with two existing studies.

The remainder of this paper is organized as follows. First, we formalize the most reliable local community search problem in Section II and develop the online search algorithm in Section III. Then, we introduce our index structure and the detailed index-based search algorithm in Section IV. The procedures of index construction, maintenance and compression are shown in Section V. After that, the experimental evaluation and results are discussed in Section VI. Finally, we discuss the related work in Section VII and conclude the work in Section VIII.

II. PRELIMINARIES AND PROBLEM DEFINITION

In this section, we first present the preliminaries about dynamic networks and then formalize the problem of the most reliable local community search.

**Definition 1** (Dynamic Networks). A dynamic network $\mathcal{G} = \{G_t, ..., G_T\}$ is a sequence of time-variant weighted graph instances $\{G_{t_1}, ..., G_{t_T}\}$ s.t., $t_1 < t_2 < ... < T$, where each timestamped instance $G_t = (V, E_t, W_t)$ contains a fixed set of nodes $V$, a set of edges $E_t$ with the weights $W_t(e) \in (0,1]$ for $\forall e \in E_t$. 

![Fig. 1. Interactions in a small graph of three timestamps.](Image)
The edge weight is a widely-used network feature to represent the interaction frequency, similarity, or connection strength between vertices. In this work, for simplicity, we normalize the edge weight to be in \( [0, 1] \).

For a graph instance \( G_t = (V, E_t, W_t) \), \( \text{deg}(u, G_t) \) denotes the degree of a vertex \( u \) in \( G_t \), which is the number of neighbors of \( u \) in \( G_t \). Like [11], we consider \( k \)-core in \( G_t \) as a connected subgraph \( G_{t,k} = (V^k, E^k) \) where each vertex has the degree no less than \( k \), i.e. \( \forall u \in V^k, \text{deg}(u, G_{t,k}) \geq k \).

**Definition 2** ((\( \theta,k \))-core). Given a graph instance \( G_t = (V, E_t, V_t) \), an integer \( k \), and a threshold \( \theta \), a connected subgraph \( G_{t,k}^\theta = (V^\theta,k, E^\theta,k, W_{t}^{\theta,k}) \) is called a \( (\theta,k) \)-core of \( G_t \) if \( G_{t,k}^\theta \) is a \( k \)-core and each edge has the weight no less than \( \theta \), i.e. \( \forall u \in V^\theta,k, \text{deg}(v,G_{t,k}^\theta) \geq k \) and \( \forall e \in E^\theta,k, W_{t}^{\theta,k}(e) \geq \theta \).

**Definition 3** (Time Interval based \((\theta,k)\)-core Reliable Community (CRC)). Given a dynamic network \( G = \{G_1,...,G_T\} \), an integer \( k \), a threshold \( \theta \), and a time interval \( T_{C} = [t_s, t_e] \), a \((\theta,k)\)-core reliable community is a subgraph \( C = (V_C, E_C) \) that spans continuously from \( t_s \) to \( t_e \) and for each timestamp \( t_n \in T_C \), the subgraph induced by \( E_C \) from the graph instance \( G_{t_n} \) is a \((\theta,k)\)-core, i.e. \( \forall v \in V_C, G_{t_n}[E_C] \) is a \((\theta,k)\)-core of \( G_{t_n} \). In the remainder of this work, \((\theta,k)\)-core reliable community is called CRC for brevity.

Based on Definition 3, a CRC maintains a cohesive structure with the required connection strength of a time interval in the dynamic network. Its reliability score can be measured by coupling the continuity and size of the community as below:

**Definition 4.** (Reliability Score of CRC) Given a dynamic network \( G = \{G_1,G_2,...,G_T\} \) and a query time interval \( T_Q = [t_i, t_j] \), the reliability score \( S_{rel}(C) \) of a CRC \( C = (V_C, E_C) \) with regard to the time interval \( T_C = [t_s, t_e] \), where \( T_C \subseteq T_Q \), is calculated as:

\[
S_{rel}(C, T_C) = \frac{|V_C| \cdot |T_C|}{|V| \cdot |T_Q|}
\]

(1)

Based on the above definitions and reliability measurement, we formalize the most reliable local community search problem as below.

**Problem 1** (Most Reliable Local Community Search). Given a dynamic network \( G = \{G_1,G_2,...,G_T\} \), a query vertex \( q \), a threshold \( \theta \), a structural constraint integer \( k \), and a query time interval \( T_Q = [t_i, t_j] \), the problem of the Most Reliable Local Community Search is to find the CRC \( C = (V_C, E_C) \) and its continuous time interval \( T_C = [t_s, t_e] \), satisfying

\[
\arg \max_{C \subseteq V} S_{rel}(C)
\]

(2)

subject to \( \forall v \in V_C, \text{deg}(v, C) \geq k; \forall e \in E_C \land \forall t \in T_C, W_t(e) \geq \theta; \) and \( T_C \subseteq T_Q \).

As shown in Figure 1 when the query time interval is \([t_1, t_3]\), we can obtain two reliable communities \( C_1 \) and \( C_2 \) w.r.t. the query input \((v_0, 0.5, 2)\). Their reliability scores are \( S_{rel}(C_1) = (5 \times 2)/(10 \times 3) = 0.33 \) and \( S_{rel}(C_2) = (4 \times 3)/(10 \times 3) = 0.4 \), respectively.

To solve the most reliable community search problem, a naive solution is to compute all the \((\theta,k)\)-core at each timestamp, and then verify their longest duration in the dynamic network. After that, their reliability scores can be obtained by multiplying their size and the number of continuous timestamps. Finally, the most reliable community can be returned by selecting the ones with the maximum reliability scores. However, the operation of finding all the \((\theta,k)\)-core needs to probe all the combinations of edges that form a connected subgraph with no less than \( k(k+1)/2 \) edges and \( k+1 \) vertices, i.e., satisfying the conditions of minimal \( k \)-core component. Thus, we can remark that the computational cost of finding the most reliable community is in exponential complexity.

III. ONLINE RELIABLE COMMUNITY SEARCH

To efficiently solve the problem of reliable local community search, in this section, we will present a novel Eligible Edge Filtering (EEF) based Online CRC Search algorithm. Different from the naive idea discussed in Section II, EEF does not need to generate all the \((\theta,k)\)-core candidates, which can greatly reduce the query time cost.

Given a graph instance \( G_{t_n} = (V, E_{t_n}) \) and a weight threshold \( \theta \), we can filter out “ineligible” edges whose weights are less than \( \theta \) and maintain only the “eligible” edges. We identify the set of eligible edges, denoted by \( E_{t_n, \theta} \), i.e. \( E_{t_n, \theta} = \{e \in E_{t_n} | W_{t_n}(e) \geq \theta \} \) for the CRC construction.

**Definition 5** (Eligible Lasting Time of Edge). Given a graph instance \( G_{t_n} = (V, E_{t_n}) \) at timestamp \( t_n \in [t_i, t_j] \), and a threshold \( \theta \), for an edge \( e \in E_{t_n, \theta} \), its eligible lasting time \( \lambda_{t_n, \theta}(e) \) is measured by the length of the longest time interval \([t_m, t_n]\) \((t_i \leq t_m \leq t_n)\) if \( W_{t_n}(e) \geq \theta \) for \( \forall t_x \in [t_m, t_n] \).

Eligible lasting time calculates the number of continuous timestamps that the edge is “eligible” until the current timestamp. For example, in Figure 1 for edge \( e = (v_0, v_1) \), we have \( \lambda_{t_1, 0.6}(e) = 1, \lambda_{t_2, 0.6}(e) = 2 \) and \( \lambda_{t_3, 0.6}(e) = 0 \). We can easily derive that the eligible lasting time of edges can be incrementally computed by accessing the dynamic network chronologically.

Based on the eligible time, we can easily identify common edges of multiple continuous graph instances, that can be utilized to construct CRC with varying duration. Different from the vertex-induced subgraph, the subgraph induced by an eligible edge set provides the guarantee to meet the requirement of edge weight \( \theta \). Therefore, the eligible edge set can be used to prune the unqualified \( k \)-core candidates by using the following property.

**Property 1** (Minimum \( k \)-core). Given an edge set \( E_{t_n, \theta} \) at timestamp \( t_n \) with regards to a threshold \( \theta \), \( E_{t_n, \theta} \) can be pruned without probing its induced communities if it satisfies \( |E_{t_n, \theta}| < k(k+1)/2 \), i.e., the number of edges does not meet the density requirement of \( k \)-core.
In addition, we can calculate the upper bound of the reliability score of CRCs constructed using $E_{t_n, \theta}$.

**Property 2** $(S_{rel}$ Upper Bound of CRC w.r.t. $E_{t_n, \theta}$). Given an eligible edge set $E_{t_n, \theta}$, an integer $k$, the upper bound reliability score $UBR_{t_n}^1$ of the CRC constructed using $E_{t_n, \theta}$ whose duration is $d$ is calculated as:

$$UBR_{t_n}^1 = 2d \cdot |E_{t_n, \theta}|/k$$

for simplicity, the normalizer $|V| \cdot |T_Q|$ is ignored.

Given $G' = (V', E')$ as a $k$-core in the induced subgraph $G_{t_n}[E_{t_n, \theta}]$, $|E_{t_n, \theta}| \geq |E'| \geq (k \cdot |V'|)/2$ must hold because there are at least $k$ edges for a vertex in the $k$-core. Therefore, if $G'$ can form a CRC with its duration as $d$, then the above inequation can be rewritten as $d \cdot |E_{t_n, \theta}| \geq (d \cdot k \cdot |V'|)/2$, which can be transformed as $d \cdot |V'| \leq 2d \cdot |E_{t_n, \theta}|/k$. Thus, we can utilize the transformed inequation to determine the upper bound of reliability score of the CRC that is constructed by the given eligible edge set.

The key idea of EEF-based Online CRC Search is to filter out edges in each graph instance $G_{t_n}$ using the given threshold $\theta$ while maintaining the last time of each edge by a timestamp. It also exploits the potential edge sets by using a search priority according to their upper bounds, which can prune the eligible edge sets with small upper bound values as much as possible. As shown in Algorithm 1, we first initialize $C_{opt}$ and $maxS$ to store the most reliable community and its reliability score (line 1). Then, for each timestamp $t_n$, we initialize a queue $Q$ and traverse the edges of $G_{t_n}$ starting from the query vertex $q$ in the Breadth-First Search manner. In this process, we update $G_{t_n}$ by removing vertices whose degree is less than $k$, and remove edges whose weight is less than $\theta$. This guarantees only to visit the vertices that are connected to $q$ by a path of eligible edges, and has no less than $k$ neighbors. During the traversing, the eligible time of edges is updated incrementally, and the eligible edges are added to the edge set $E_{t_n, \theta}$. After $E_{t_n, \theta}$ is constructed, we calculate its upper bound $UBR_{t_n}^1$ of potential CRC whose duration is 1 (lines 2-19). Then, we visit each timestamp $t_n$ in the descending order of $UBR_{t_n}^1$ which provides a best-first search strategy to exploit the CRCs. We utilize $E_{t_n, \theta}$ to construct CRC with duration $d$ iterating from 1 to $|[t_1, t_n]|$ (lines 20-28). At each iteration, we select the edge set $E'$ where each edge has the eligible time no less than $d$ and update the upper bound w.r.t. $d$. Then we adopt Property 1 and Property 2 to prune the CRC construction if $|E'|$ is too small or $UBR_{t_n}^1$ cannot exceed $maxS$. After that, we can extract the CRC $C$ (i.e., local maximal $k$-core) from the induced subgraph $G_{t_n}[E']$ by finding the connected component containing $q$ after the core decomposition, and then update $C_{opt}$ and $maxS$. Finally, the algorithm returns $C_{opt}$ as the optimal result.

The time complexity of Algorithm 1 can be analyzed as below. For each graph instance $G_{t_n} = (V, E_{t_n})$, it takes $O(|V| + |E_{t_n}|)$ to run the Breadth-First Search that requires to visit very vertex and edge once. At the same time, the eligible time of each edge is obtained (lines 2-18). Then, it needs $O(\Theta(t_n, |E_{t_n}|))$ to compute $|[t_1, t_n]|$ number of CRCs where each CRC is obtained by a core decomposition process that needs to compute $O(|E_{t_n}|)$ (12) (lines 20-27). Therefore, for the query interval of $|T_Q|$ timestamps, Algorithm 1 takes $O\left(\sum_{t_n\in T_Q}(\Theta(|V| + |E_{t_n}|) + \Theta(|t_1, t_n| \cdot |E_{t_n}|))\right)$ in total, which can be rewritten as $O(T_Q \cdot |E|_t + |T_Q| \cdot |E|_t)$, i.e., $O(|T_Q| \cdot |E|_t + |T_Q| \cdot |E|_t)$, where $|E|_t$ denotes the average number of edges of the graph instances.

### IV. INDEX BASED RELIABLE COMMUNITY SEARCH

To further accelerate the query processing, in this section, we first propose a forest index structure, called Weighted Core Forest Index (WCF-Index), to maintain the $(\theta, k)$-core vertices for each graph instance $G_{t_n}$. Then, we develop an index-based dynamic programming algorithm by using the proposed index and deriving the upper bound of reliable communities.
A. WCF-Index

The general idea of this index is to maintain the vertex candidates of the \((\theta, k)\)-core with regards to the given \(\theta\) and \(k\) at each timestamp, from which we can work out the satisfied CRCs containing \(q\) with the different continuous time intervals.

**Definition 6** (\(\theta\)-threshold of a Vertex). Given a graph instance \(G_t=(V, E_t, W_t)\) at a timestamp \(t\) and an integer \(k\), for a vertex \(u \in V\), it may have a set of \(\theta\) values and their corresponding \((\theta, k)\)-core subgraphs containing \(u\). Thus, we take the largest \(\theta\) value in the \(\theta\)-set as the \(\theta\)-threshold of \(u\), denoted as \(\theta\)-thres\(_k\)(\(u, G_t\)).

**Example 2.** Figure 2 shows the \(\theta\)-threshold of vertices in \(G_t\), in Figure 1 (a) with regards to different \(k\) values, e.g., \(\theta\)-thres\(_2\)(\(v_1, G_t\)) is 0.5 because \((0.5, 2)\)-core (i.e., \(\{v_0, v_1, v_2, v_3, v_4\}\) exists in \(G_t\), but no one \((\theta', 2)\)-core containing \(v_1\) exists if \(\theta' > 0.5\).

According to the above definition and the example, we are able to justify whether a vertex \(v\) is contained in a \((\theta, k)\)-core for given \(\theta\) and \(k\) if the \(\theta\)-threshold values of vertices are maintained. However, the \(\theta\)-threshold only implies the vertex candidates of a \((\theta, k)\)-core, but fails to reflect the structural connectivity of vertices. Therefore, it is highly desirable to design an index structure for maintaining the \(\theta\)-threshold and the structure information of vertices together.

Yang et al. in [13] proposed a forest-based index to query \((k, \eta)\)-core in a static uncertain graph where \(k\) implies the degree constraint of the vertex and \(\eta\) implies the probability of the vertex to appears in the subgraph. By maintaining \(\eta\)-tree\(_k\) for each \(k\), it can accelerate the search of all the \((k, \eta)\)-core with custom \(\eta\) requirements. Motivated by \(\eta\)-tree\(_k\), in this work, we extend the concept of \(\eta\)-tree\(_k\) to the dynamic weighted network to construct the \(\theta\)-tree\(_{k,t}\) for each \(k\) at time \(t\), which can support quick retrieval of local maximal \((\theta, k)\)-core from the indexed graph instance \(G_t\).

**Definition 7** \((\theta\)-tree\(_{k,t}\)). Given a graph instance \(G_t\), an integer \(k\), \(\theta\)-tree\(_{k,t}\) index is a tree structure, satisfying

1) **Node**: each tree node \(V\) is a set of maximal connected vertices in \(G_t\) with same \(\theta\)-threshold value, denoted as \(\forall \theta, \forall v \in V, \theta\)-thres\(_k\)(\(v, G_t\)) = \(\forall \theta\);

2) **Parent-child relationship**: if a node \(V\) is the parent of a node \(W\), then we get that \(V\) and \(W\) are connected in \(G_t\), and \(\forall \theta < \forall W, \theta\).

**Example 3.** Figure 3 presents the constructed \((\theta\)-tree\(_{2,t}\) of \(G_t\), from \(\theta\)-threshold of \(G_t\), in Figure 2. If we search \((0.5, 2)\)-core on \(\theta\)-tree\(_{2,t}\), three tree nodes will be returned, i.e., \(\{v_0, v_1\}, \{v_2, v_3, v_4\}\), and \(\{v_7, v_8, v_9\}\). These tree nodes can induce two \((0.5, 2)\)-core, i.e., \(g_1\) and \(g_2\).

**Remark**: \(\theta\)-tree can be composed of several trees where each tree represents a connected component in the graph instance. The WCF-Index is composed of \(\theta\)-tree\(_k\) of each \(k\) and \(t\) in the dynamic network. In addition, we set the \(\theta\)-threshold as the standard values \(\{0.0, 0.1, 0.2, \ldots, 0.9, 1.0\}\). If the \(\theta\)-threshold of a vertex is not in the standard set, we will round it down to the nearest standard value. However, users can specify any \(\theta\) value in query processing and the index will take the nearest round down standard value. For instance, given a query with \(\theta = 0.55\), the index accesses the tree nodes from 0.5 and then examines the \(\theta\)-threshold of vertices in the tree node \(V\) if \(\forall \theta < 0.55\). In the following discussion, we skip this process for simplicity.

B. Dynamic Programming based CRC search

To solve the most reliable community search problem, we need to compare CRC with different duration. In this section, we develop a dynamic programming algorithm based on the recursive relation of CRCs ending in consecutive timestamps and utilize the WCF-Index to search CRC with varying duration efficiently.

Assume that the lasting time interval of the CRC is fixed (so does the duration), then we only need to extract the CRC with the largest size. Given the duration of the CRC is \(d\) and the last timestamp it spans is \(t_n\), we denote the maximal CRC w.r.t. the query input as \(C(d, t_n)\). We can easily derive the following recursive relation between CRCs:

\[
C(d, t_n) \subseteq C(d - 1, t_{n-1}) \cap C(d - 1, t_n) \quad (4)
\]

The base situation is \(C(1, \cdot)\) that can be retrieved from WCF-Index. Based on Eq. (4) we can devise a DP algorithm to compute \(C(d, t_n)\). More specifically, to get \(C(d, t_n)\), we simply compute the intersection of \(C(d - 1, t_{n-1})\) and \(C(d - 1, t_n)\) and extract the local maximal \((\theta, k)\)-core using core decomposition. The intermediate result of \(C(d, t_n)\) with varying \(d\) is maintained to support the adoption of the dynamic programming.

If at a timestamp \(t_a\), the maximal community \(C(1, t_a)\) does not exist, i.e. for a given query, there is no such subgraph
satisfying \((\theta, k)\)-core constraint at time \(t_a\), then it implies that further calculations depending on \(C(1, t_a)\) are unnecessary. In this work, these kinds of timestamps like \(t_a\) are called anchored timestamps of a query time interval. The anchored timestamps split the query interval \([t_i, t_j]\) into several non-overlapping time intervals \(T_S = \{T_1, T_2, \ldots\}\). For each interval \(T_i \in T_S\), we can compute the upper bound of the reliability score of the communities.

**Property 3 (\(S_{rel}\) Upper Bound of CRC w.r.t. \(T_i\)).** Given a time interval \(T_i = [t_s, t_e]\) where \(C(1, t_a)\) exists for every \(t_n \in [t_s, t_e]\), we can construct an array \(M = (\mu_1, \mu_2 + 1, \ldots, \mu_n)\) to store the size of \(C(1, \cdot)\), where \(\mu_n\) denotes the size of \(C(1, t_n)\). The upper bound reliability score (UBR) in this time interval can be calculated by:

\[
UBR_{T_i} = \max_{\mu_n \in M} (\mu_n \ast LCT(\mu_n, M)) \quad (5)
\]

where \(LCT(\mu_n, M)\) stands for the length of the Longest Consecutive Timestamps of \(\mu_n\) in array \(M\). For simplicity, the normalizer \(|V| \cdot |T_Q|\) is ignored.

**Example 4.** Figure 4 shows an illustrative example of calculating the UBR of interval \([t_1, t_2]\). In \([t_1, t_2]\), the largest possible reliability score is obtained by a CRC that contains three vertices and spans for five timestamps.

**Remark:** Similar to Property 2, we use \(UBR^d_{T_i}\) to denote the upper bound calculated by the size of \(C(d, \cdot)\). However, it can determine the maximum reliability score of the community whose duration is no less than \(d\). In the process of community search, \(UBR^d_{T_i}\) can be updated with different duration of CRC and provide sustainable pruning power. Having \(UBR^d_{T_i}\) calculated for each interval \(T_i\) and updated during the community exploration, we can skip exploring CRCs if \(UBR^d_{T_i}\) is no larger than the reliability score of intermediate community candidates we have obtained during the query processing.

Algorithm 2 presents the detailed dynamic programming procedure of the WCF-Index based CRC Search. We first initialize a table \(L_{\cdot}^d, \max S,\) and \(C_{\cdot}^d\) to store the extracted communities, the maximum reliability score, and the most reliable community (line 1). For each timestamp \(t_n \in [t_i, t_j]\), we can obtain \(C(1, t_n)\) from the \(\theta\)-tree, and store it in \(L_{\cdot}^1[n]\) (lines 2-6). In addition, we also determine whether \(t_n\) is an anchored timestamp. Then we split \(T_Q\) into several non-overlapping time intervals \(T_S = \{T_1, T_2, \ldots\}\) with valid \(C(1, \cdot)\) by the anchored timestamps, and calculate their upper bound reliability score (lines 7-8). For each individual time interval \(T_i = [t_s, t_e]\), if its upper bound is no larger than \(\max S\), then the time interval is pruned (line 10). Otherwise, we initialize an array \(Q\) to store the size of CRC and compute the CRC with various duration \(d\) based on Eq.\(d\) and store the intermediate CRC \(C(d, t_x)\) in \(L_{\cdot}^d[x]\) (lines 11-16). Then, we update \(\max S\) and \(C_{\cdot}^d\) and add the size of \(C(d, t_x)\) to \(Q\) for upper bound calculation (lines 17-20). Once \(C(d, \cdot)\) has been explored for every \(t_x \in [t_s, t_e]\), we can update the \(UBR^d\) and determine whether it is necessary to explore communities with longer duration in this interval (line 21-22). Finally, the algorithm returns the most reliable community \(C_{\cdot}^d\) whose reliability score is the largest.

**Algorithm 2: WCF-Index based CRC Search**

**Input:** A dynamic weighted graph \(G = \{G_{t_1}, G_{t_2}, \ldots, G_{t_T}\}\), query time \([t_i, t_j]\), the WCF-Index, integer \(k\), weight threshold \(\theta\), query vertex \(q\)

**Output:** the most reliable community \(C_{\cdot}^d\)

1. \(L_{\cdot}^d \leftarrow [\emptyset]; \max S \leftarrow 0; C_{\cdot}^d \leftarrow \emptyset;\)
2. for \(t_n \in [t_i, t_j]\) do
3. \(\text{Extract } C(1, t_n) \text{ from WCF-Index}[k][t_n];\)
4. if \(C(1, t_n) = \emptyset\) then
5. \(\text{set } t_n \text{ as anchored timestamp;}\)
6. else \(L_{\cdot}^1[n] \leftarrow C(1, t_n);\)
7. Get all the consecutive timestamps \(T_S = \{T_1, T_2, \ldots\}\) split by the anchored timestamp;
8. Calculate upper bound \(ub = \{UBR^d_{T_1}, UBR^d_{T_2}, \ldots\}\) for each consecutive time sequence;
9. for \(T_i = [t_s, t_e] \in T_S\) by descending of \(UBR^d_{T_i}\) do
10. if \(UBR^d_{T_i} \leq \max S\) then continue;
11. for \(d \leftarrow 1\) to \([t_s, t_e]\) do
12. \(Q \leftarrow \emptyset;\)
13. for \(t_x \in [t_s, t_e]\) do
14. if \(d \leq |[t_s, t_e]|\) then
15. \(\text{if } d > 1\) then
16. \(L_{\cdot}^d[x] \leftarrow \text{local maximal } k\)-core in \(L_{\cdot}^{d-1}[x-1] \cap L_{\cdot}^{d-1}[t_x];\)
17. \(\max S \leftarrow \max(\max(\max S_{rel}(L_{\cdot}^d[x])), \max S);\)
18. if \(S_{rel}(L_{\cdot}^d[x]) \geq S_{rel}(C_{\cdot}^d)\) then
19. \(C_{\cdot}^d \leftarrow L_{\cdot}^d[x];\)
20. \(Q.\text{append}(L_{\cdot}^d[x]);\)
21. \(UBR^d_{T_i} \leftarrow \max_{q \in Q}(q \ast LCT(q, Q));\)
22. if \(UBR^d_{T_i} \leq \max S\) then break;
23. Return \(C_{\cdot}^d\)

The time complexity of Algorithm 2 is dominated by the operation of finding CRCs with various duration (lines 10-23) as the \(C(1, \cdot)\) community can be queried from the index in constant time. In the worst case, there are up to \(|T_Q|^2\) subgraphs to be explored and the community construction takes \(O(|E_i|)\) complexity, where \(T_Q\) is the query interval and
$|E_t|$ is the average number of edges of graph instance $G_t$. The total complexity is $O(|T_Q|^2 \cdot |E_t|)$. Compared with the EEF-based Online CRC Search Algorithm, WCF-Index can avoid searching a large number of edges, which helps to reduce the time cost of computing $C(1, \cdot)$.

V. WCF INDEX CONSTRUCTION, MAINTENANCE, AND COMPRESSION

In this section, we describe the procedure of index construction, maintenance, and compression, which can support efficient query processing over dynamic weighted networks.

A. WCF-Index Construction

The main idea of constructing WCF-Index is to build the $\theta$-tree$_{kt}$ for each graph instance $G_t$ for $k \in [1, \kmax]$, where $\kmax$ denotes the maximum core number of the vertex in $G_t$. To obtain the $\theta$-tree$_{kt}$, we group the vertices by their $\theta$-threshold value and add the vertex groups as tree nodes into the $\theta$-tree according to the $\theta$-threshold and connectivity of the tree nodes, i.e., the vertex groups.

Algorithm 3 presents the procedure of building the WCF-Index. For each graph instance $G_t$, we build $\theta$-tree$_{kt}$ for each available $k$ by first computing $\theta$-threshold of vertices and then construct and insert tree nodes to the $\theta$-tree index. We first initialize two graphs $G_{pre}$ and $G_{cur}$ to store intermediate states of edge filtering (line 3). Then we iteratively pick $\theta' \in \Theta$ in descending order. For each $\theta'$, we build the edge set $E_{t, \theta'}$ whose weights are no less than $\theta'$ and obtain the induced subgraph $G_t[E_{t, \theta'}]$ as the current state $G_{cur}$. We can obtain a set of vertices $V_{\theta'}$ whose core number in $G_{cur}$ is increased with regards to the core number in the last state $G_{pre}$. This implies that for a vertex $w \in V_{\theta'}$, the $\theta$-threshold of $w$ is $\theta'$ w.r.t. its increased core number (lines 4-7). After that, we set the previous state $G_{pre}$ to be $G_{cur}$ and obtain the distinct values of the newly increased core numbers $K$ (lines 8-9). Then, according to the newly identified $\theta$-threshold, we can construct and add tree nodes to the $\theta$-tree$_{kt}$ for $k' \in K$ (lines 10-23). To do that, we get $\theta$-tree$_{kt}$ from the WCF-Index, then we identify the sets of connected vertices whose $\theta$-threshold at $k'$ is $\theta'$ as the tree node $X$ (lines 11-14). To determine the position of $X$, we find each tree node $Y$ that contains any neighbor $v$ of $G_t[X]$ and its root $Z$, so that $X, Y, Z$ are connected (lines 15-18). If $Z, \theta > \Theta, \theta$, $X$ is assigned as the parent of $Z$, otherwise, their $\theta$-threshold are the same because the smaller $\theta'$ has not been visited yet, so we need to merge $X$ to $Z$ (lines 19-21). After iterating all the standard threshold values, we can construct all the $\theta$-tree$_{kt}$ completely for each possible $k$ of each $G_t$ and return the WCF-Index.

Example 5. Consider the graph instance $G_t$ in Figure 4, we calculate the $\theta$-threshold of its vertices by inducing $G_t[E_{t, \theta}]$ with increasing $\theta$. Upon inducing $G_t[E_{t, 0.5}]$, comparing to $G_t[E_{t, 0.6}]$, we can observe the core number increase of $v_0$ and $v_1$ from 1 to 2, which implies $\theta$-thres$_2(v_0, G_t) = \theta$-thres$_2(v_1, G_t) = 0.5$. So we add tree node to $\theta$-tree$_2$ because the core number is increased to 2. One node $X = \{v_0, v_1\}$ is constructed as $v_0$ and $v_1$ are connected. $\theta$-tree$_{2, t_1}$ has two nodes $Y_1 = \{v_2, v_3, v_4\}$ and $Y_2 = \{v_5, v_6, v_9\}$ from previous steps and the node that contains neighbors of $X$ is $Y_1$, whose root is itself. $X$ can thus be added as the parent of $Y_1$ since $Y_1, \theta > X, \theta$. By now, $\theta$-tree$_{2}$ contains three tree nodes. After inducing $G_t[E_{t, 0}]$ we can construct $\theta$-tree$_{2, t_1}$ as Figure 5, $\theta$-tree$_{1, t_1}$ and $\theta$-tree$_{3, t_1}$ will also be obtained.

B. WCF-Index Maintenance

It is costly to compute the entire $\theta$-tree for each timestamp in the dynamic network. In the real world, the weight between vertices tends to change progressively, so the difference between snapshot graphs can be subtle. Rather than reconstructing the index for each timestamp from the scratch, we can construct the index by updating operations based on the change of the graph instances. Therefore, we need to develop an index maintenance strategy to accelerate the index construction.

Batagelj et al. in [12] proposed an incremental core number update method for an evolving graph, which locates a small set of vertices whose core number will be affected by using the below two concepts.

Definition 8 (Subcore in [12]). Given a graph $G=(V, E)$ and a vertex $u \in V$, the Subcore of $u$ denoted as $S_u$, is a set of vertices that has the same core number as $u$ and is connected
with $u$ via a path, where each vertex on the path has the same core number as $u$.

**Definition 9** (Purecore in [12]). Given a graph $G=(V,E)$ and a vertex $u \in V$, the Purecore of $u$ denoted as $P_u$, is a set of vertices where each vertex $v \in P_u$ satisfies:

1) Condition 1: the core number $\text{core}(w, G)$ of $w$ is equal to the core number $\text{core}(u, G)$ of $u$.
2) Condition 2: $w$ has a set $W$ of neighbors whose core numbers are no less than $\text{core}(w, G)$, and $|W|$ is larger than $\text{core}(u, G)$.
3) Condition 3: $w$ is connected to $u$ via a path, where each vertex on the path satisfies the conditions (1) and (2).

Specifically, given two vertices $u$ and $v$ in a graph $G=(V,E)$, and $\text{core}(u, G) \leq \text{core}(v, G)$, if an edge $(u, v)$ is removed from $G$, then only the vertices in the purecore set $P_u$ may have their core number increased. We can extend the rules to the weighted graph, in which the updates include edge insertion, edge deletion, and edge weight change.

Considering that $G_{t_1}$ is obtained by inserting an edge $(u, v)$ with weight $\theta'$ to $G_{t_1} = (V, E_{t_1}, W_{t_1})$, for a vertex $w \in V$, if $\theta \text{thres}_k(w, G_{t_1}) = \theta''$ and $\theta'' > \theta'$, then $\theta \text{thres}_k(w, G_{t_2})$ will remain unchanged.

**Property 4** (Insertion of an Edge). Given a graph $G_{t_1}$ and $\theta \text{tree}_{k, t_1}$, for each $k \in [1, k_{\text{max}}]$, and two vertices $u$ and $v$ such that $\theta \text{thres}_k(u, G_{t_1}) \leq \theta \text{thres}_k(v, G_{t_2})$, if an edge $(u, v)$ is inserted with weight $\theta'$, then only the vertices $\{w \in P_u | \theta \text{thres}_k(w, G_{t_1}) < \theta\}$ may have their $\theta$-threshold increased.

**Property 5** (Deletion of an Edge). Given a graph $G_{t_1}$ and $\theta \text{tree}_{k, t_1}$, for each $k \in [1, k_{\text{max}}]$, and two vertices $u$ and $v$ such that $\theta \text{thres}_k(u, G_{t_1}) \leq \theta \text{thres}_k(v, G_{t_2})$, if an edge $(u, v)$ is removed with weight $\theta'$, then only the vertices $\{w \in S_u | \theta \text{thres}_k(w, G_{t_1}) \leq \theta\}$ may have their $\theta$-threshold decreased.

The edge insertion and edge deletion can be treated as the update of edge weight.

**Property 6** (Update of Edge Weight). Given a graph $G_{t_1}$ and $\theta \text{tree}_{k, t_1}$, for each $k \in [1, k_{\text{max}}]$, two vertices $u$ and $v$ such that $\theta \text{thres}_k(u, G_{t_1}) \leq \theta \text{thres}_k(v, G_{t_2})$, we have $\theta$ if the weight of edge $(u, v)$ increases from $\theta_1$ to $\theta_2$ (w.r.t. $\theta_1 < \theta_2$), then only the vertices $\{w \in P_u | \theta \text{thres}_k(w, G_{t_1}) \in (\theta_1, \theta_2)\}$ may have their $\theta$-threshold increased; (2) if the weight of edge $(u, v)$ decreases from $\theta_2$ to $\theta_1$ (w.r.t. $\theta_1 < \theta_2$), then only the vertices $\{w \in S_u | \theta \text{thres}_k(w, G_{t_1}) \in (\theta_1, \theta_2)\}$ may have their $\theta$-threshold decreased.

**Example 6.** Figure 5(a) shows an updated graph instance of $G_{t_1}$ by adding an edge $(v_3, v_5)$ with weight 0.3 to $G_{t_1}$. We can identify $P_{v_3} = \{v_0, v_2, v_3, v_5, v_6, v_7, v_8\}$ in $G_{t_1}$. In addition, the $\theta$-threshold of $v_3$ or $v_5$ is less than 0.3 and other vertices’ $\theta$-threshold is no less than 0.3. According to Property 5, only $v_5$ and $v_6$ may have their $\theta$-threshold increased. After recalculating $\theta \text{thres}_2(v_5, G_{t_1})$ and $\theta \text{thres}_2(v_6, G_{t_1})$, we can update the tree index from Figure 5(a) to Figure 5(b).

### C. WCF-Index Compression

Sometimes, the graph instances of some consecutive timestamps may not have a big change, which leads to much duplicate information across $\theta$-tree indices. Thus, the redundancy can be reduced based on the following observations. We have the following observations on the $\theta$-tree

**Observation 1.** The weight of the edge changes progressively over time and the graphs in consecutive timestamps have a similar structure.

**Observation 2.** In the $\theta$-tree, a tree node usually has multiple vertices.

Therefore, our compression strategy is to construct a virtual node to replace the tree node that contains multiple vertices and appears frequently. The actual vertices of the virtual node are stored in an auxiliary table.

**Example 7.** Figure 6 shows an example of compressed $\theta$-tree, and $\theta$-tree, where the nodes with vertices $\{v_0, v_1\}$, $\{v_2, v_3, v_4\}$, $\{v_7, v_8, v_9\}$ are replaced by virtual nodes $X_1$, $X_2$, $X_3$.

Each virtual node can be regarded as an encoding of a unique vertex set. The space cost can be reduced if we use an auxiliary table to only maintain one copy of the tree nodes that frequently appear in WCF-Index, and keep a virtual id at the positions of these nodes in WCF-Index. To make the
compression, we only select the tree nodes that can bring positive space gain.

**Definition 10 (Space Gain).** The space gain is defined as the space saved from replacing the tree node \(X\) as a virtual node \(X\), that is

\[
SG(X) = f \cdot (|X| - 1) - |X|
\]

where \(f\) is the frequency of \(X\) that appears in the WCF-Index, and \(|X|\) is the size the vertex set.

By scanning and counting the frequency of tree nodes, we can calculate the space gains for all the tree nodes, and generate the compressed index easily. We do not provide the pseud codes in this paper due to the limited space.

**VI. EXPERIMENT**

We conduct extensive experiments to evaluate the performance of our proposed algorithms, including EEF-based Online CRC Search in Algorithm 1 WCF-Index based CRC Search in Algorithm 2 and \(\theta\)-Tree Construction in Algorithm 3 denoted as EEF-CRC, WCF-CRC and WCF-Construct, respectively. We also evaluate index maintenance in Section V-B and index compression in Section V-C denoted as WCF-Maintain and WCF-Compress. All the experiments are conducted on a Windows machine with an Intel i9-10900F CPU @ 2.80GHz and 32.0 GB DDR4-RAM.

**A. Experimental Setup**

**Datasets.** We conduct the experiments on eight real-world dynamic network datasets collected from SNAP\(^1\) and Network Data Repository\(^2\). In BitcoinAlpha (BA) and BitcoinOTC (BO) datasets, the edge weight represents the rating between two users. In the remaining datasets, the edge weight is calculated from the interaction frequency. The edge weight of all the datasets is normalized to \([0, 1]\) by min-max normalization. The statistics of the dataset are shown in Table I. The number of vertices and edges are denoted as \(|V|\) and \(|E|\), respectively. Each dataset is equally divided into \(|T|\) graph instances depending on the size of the dataset. For example, the largest dataset StackOverflow (SOF) is divided into \(|T| = 100\) snapshots and the medium-sized datasets, e.g., TechAsTopology (TAT), Retweet, etc. are divided into \(|T| = 30\) instances. We denote \(|\hat{V}|\), \(|\hat{E}|\), density and \(\hat{k}_{max}\) as the average of vertex numbers, edge numbers, density, and the largest core numbers of the \(|T|\) graph instances, respectively.

**Parameters.** Table II shows the detailed setting of the parameters used in the experiments. We vary the query parameter \(k\) as 20\%, 40\%, 60\%, 80\% of \(k_{max}\) for each dataset with the default value 40\%. The threshold value varies from 0.0 to 0.8 with the default value of 0.4. The length of the query time interval was specified as 2, 5, 10, 15, 20 with the default value of 10. Their default values are marked in bold font. We sample 100 query vertices whose core numbers are uniformly distributed in \([1, k_{max}]\) for each dataset and report their average running time as the time cost. The average core number of the 100 query vertices is shown in Table II as \(\hat{k}_{query}\). In general, \(\hat{k}_{query}\) is around 50\% of \(k_{max}\), which reflects the common scenario of query vertex.

**B. Efficiency Evaluation of Query Processing**

In this section, we present the performance of our proposed algorithms under the default parameter settings. Figure 7 demonstrates the time cost when we run the two algorithms over the eight datasets. WCF-CRC runs much faster than EEF-CRC for every dataset. For instance, WCF-CRC reduces the time cost of EEF-CRC by about 89 times.

To show the impact of each parameter, we also evaluate the efficiency of the proposed algorithms by varying the values of parameters \(k\), \(\theta\), and \(t\), respectively. We utilize four representative largest datasets Email, Reddit, HepPh, SOF to demonstrate the experimental results.

**Varying \(k\).** Figure 8 shows the average time cost of our proposed algorithms when \(k\) varies from 20\% to 80\% of the corresponding \(k_{max}\) value in four datasets. With the increase of \(k\), both EEF-CRC and WCF-CRC queries consume decreasing time in the similar trend. And EEF-CRC consumes significantly higher time than WCF-CRC for all datasets. For instance, EEF-CRC takes 3.37s, but WCF-CRC only needs 0.035s to complete the query processing in Reddit dataset where \(k\) is 40\% of the average large core number (i.e., \(k = 5\)). The gap between two algorithms in Email is smaller than that in other datasets. This is because Email has a particularly

---

1 https://snap.stanford.edu/data/
2 https://networkrepository.com
higher density than other datasets, which leads to less number of vertices that can be pruned by using the WCF-Index.

Varying threshold \( \theta \), Figure 9 shows the average time cost of our proposed algorithms when the threshold \( \theta \) varies from 0 to 0.8. WCF-CRC outperforms EEF-CRC significantly. For instance, in HepPh, when \( \theta = 0 \), EEF-CRC takes 17.26s while WCF-CRC takes 1.98s. WCF-CRC is faster than EEF-CRC by about 9 times. When \( \theta \) is given as 0.8, the two algorithms take 16.0s and 0.39s, respectively, i.e., the acceleration of WCF-CRC than EEF-CRC is about 40 times. With the increase of \( \theta \), the speedup trend of WCF-CRC is obvious because the WCF-Index returns less tree node when \( \theta \) is large, i.e., more vertices can be pruned. The running time of WCF-CRC is insensitive to the increase of \( \theta \) because it requires scanning all the edges in the locally connected subgraph for each query vertex.

Varying time span \(|T_\theta|\), Figure 10 shows the average time cost of the proposed algorithms when the time span \(|T_\theta|\) varies from 2 to 20, respectively. Both EEF-CRC and WCF-CRC consume higher time cost when the query time span \(|T_\theta|\) increases. For instance, in the Reddit dataset, EEF-CRC takes 0.18s, 0.55s, 1.41s, 2.56s and 4.10s to complete the queries, respectively. However, WCF-CRC only takes 0.0052s, 0.0087s, 0.0184s, 0.0232s and 0.026s at the same settings. The gap between the two algorithms is increasing with the growing length of the query interval \(|T_\theta|\) because it has more graph instances to be processed and WCF-CRC is faster than EEF-CRC to get \((\theta,k)\)-core at each graph instance.

**C. Evaluation of Index Construction and Maintenance**

In Figure 11 we report the running time of WCF-Construct for all the graph instances of eight datasets. Generally, it takes around 10s for datasets with small number of vertices like BA, BO, and Email. For the largest dataset like SOF, it needs 25h to complete the index construction.

We also evaluate the effectiveness of the index maintenance method. Taking the first graph instance of Reddit as the base, we randomly sample 100, 200, 300, 500, 1000 edges and mix the operation of edge insertion, deletion, and weight update to generate a synthetic instance. Figure 12 shows the time cost of WCF-Construct and WCF-Maintain on the synthetic graph, where the speed up of WCF-Maintain is significant. For instance, reconstructing the index takes 8s, but WCF-Maintain only takes 4s when 1000 edges are updated.

**D. Evaluation of Scalability**

We evaluate the scalability of proposed algorithms including EEF-CRC, WCF-CRC and WCF-Construct by using five graph instances from two datasets Reddit and HepPh. For each
dataset, we generate four new datasets with different sizes by randomly sampling 20%, 40%, 60%, 80% edges from the dataset, respectively. The dataset itself is considered with the 100% data size. Figure 13 shows the time cost of WCF-CRC and EEF-CRC on the size-varying datasets. With the increase of the data size, we can find that the running time of WCF-CRC and EEF-CRC grow in a gentle trend, which implies that both algorithms are easily applicable to large-scale networks.

Furthermore, we also show the scalability of index construction in Figure 14. From this, we can observe a linear increasing trend of the construction time. For instance, the construction time of Reddit is 4.7s, 12.8s, 22.3s, 34.0s and 47.9s when the size of the dataset increases as 20%, 40%, 60%, 80% and 100%, respectively.

E. Evaluation of Index Size with Compression

We show the WCF-Index size of the eight datasets in Table III. For each dataset, we take ten graph instances. The largest dataset StackOverflow takes 874,931kb and the smallest dataset BitcoinAlpha takes 333kb. The compressed size is the sum of the compressed index and the auxiliary table. A significant compression effectiveness can be observed in Table III e.g., the index size of HepPh can be compressed to 50% of the original size.

F. Effectiveness Evaluation of Query Processing

To show the effectiveness of finding communities in dynamic or temporal networks, we compare our $\{(\theta,k)\}$-core reliable community (CRC) with the Persistent Community (PC) and Stable Community (SC) proposed by Li [9] and Qin [10], respectively. To do this, we select five graph instances of Reddit dataset and return the largest community $C$ obtained by SC, PC, and our CRC. To show the quality of returned communities, we utilize three community quality metrics:

- **Average Snapshot Density (ASD)** measures how dense is the community and captures the intuition that a good community should be closely connected inside. The larger is the density, the closer the community is connected. Average snapshot density is calculated as the average density of the community in each snapshot: $ASD = \sum_{i=1}^{t} \frac{density(G_t[C])}{t}$.

- **Average Snapshot Core (ASCore)** captures the degree information of vertices and evaluates the closeness of the community. The larger is the core number, the more interactions each vertex will keep with others in the community. ACore calculates the average value of the average core number of each vertex in each snapshot: $ASCore = \sum_{v \in V} \frac{\sum_{i=1}^{t} core(v,G_t[C])}{|V|}/t$.

- **Average Snapshot Conductance (ASCond)** measures how “well-knit” the graph is. The higher is the conductance, the easier the community can communicate with the vertices outside the community. In the local community detection task, the smaller conductance is desired as it implies the community is tightly self-capulated. Here, the average snapshot conductance is calculated as the average conductance of the community in each snapshot: $ASD = \sum_{i=1}^{t} \frac{conduction(G_t[C])}{t}$.

Table IV shows the experimental results of evaluating the community quality on the Reddit dataset. The community is obtained with the same structural cohesiveness constraint (core number or number of neighbors equals 8). We vary the duration or frequency of the community ($\tau$ in SC and PC, $d$ in CRC), denoted by $t$ in Table IV to compare the community quality with different temporal features. The size of each community is also provided as the average snapshot size ASS in Table IV. It can be observed that CRC performs...
In this paper, we first discussed the necessity of reliable local community in dynamic networks and proposed the novel most reliable community search problem. Then, we developed an online \((\theta, k)\)-core reliable community search approach by pruning the ineligible edges based on the given threshold and their lasting times. After that, we designed an effective \textit{WCF-Index} to maintain the vertex candidates of \((\theta, k)\)-core subgraphs, and developed an efficient index-based dynamic programming approach. Finally, the empirical evaluations on a variety of datasets and parameter settings illustrate the efficiency and effectiveness of the proposed approaches.
This figure "fig1.png" is available in "png" format from:

http://arxiv.org/ps/2202.01525v1